Q146：LRU 缓存机制

1. HashMap + 双向链表。

```
 class LRUCache {
     class DLinkedNode {
         int key;
         int value;
         DLinkedNode prev;
         DLinkedNode next;
         public DLinkedNode() {}
         public DLinkedNode(int _key, int _value) {key = _key; value =
_value;}
     }

     private Map<Integer, DLinkedNode> cache = new HashMap<Integer,
DLinkedNode>();
     private int size;
     private int capacity;
     private DLinkedNode head, tail;

     public LRUCache(int capacity) {
         this.size = 0;
         this.capacity = capacity;
         // 使用伪头部和伪尾部节点
         head = new DLinkedNode();
         tail = new DLinkedNode();
         head.next = tail;
         tail.prev = head;
     }

     public int get(int key) {
         DLinkedNode node = cache.get(key);
         if (node == null) {
             return -1;
         }
         // 如果 key 存在，先通过哈希表定位，再移到头部
         moveToHead(node);
         return node.value;
     }

     public void put(int key, int value) {
         DLinkedNode node = cache.get(key);
         if (node == null) {
             // 如果 key 不存在，创建一个新的节点
             DLinkedNode newNode = new DLinkedNode(key, value);
             // 添加进哈希表
             cache.put(key, newNode);
             // 添加至双向链表的头部
```

```
                addToHead(newNode);
                ++size;
                if (size > capacity) {
                    // 如果超出容量，删除双向链表的尾部节点
                    DLinkedNode tail = removeTail();
                    // 删除哈希表中对应的项
                    cache.remove(tail.key);
                    --size;
                }
            }
            else {
                // 如果 key 存在，先通过哈希表定位，再修改 value，并移到头部
                node.value = value;
                moveToHead(node);
            }
        }

        private void addToHead(DLinkedNode node) {
            node.prev = head;
            node.next = head.next;
            head.next.prev = node;
            head.next = node;
        }

        private void removeNode(DLinkedNode node) {
            node.prev.next = node.next;
            node.next.prev = node.prev;
        }

        private void moveToHead(DLinkedNode node) {
            removeNode(node);
            addToHead(node);
        }

        private DLinkedNode removeTail() {
            DLinkedNode res = tail.prev;
            removeNode(res);
            return res;
        }
    }
```

2. 由LinkedHashMap实现

    1. HashMap提供了3个空的方法，用于自类进行扩展。HashMap中调用了空的方法。

```
    // 将结点插入到尾部。
    void afterNodeAccess(Node<K,V> p) { }
    // 若超出容量则删除第一个结点(头结点)。
    void afterNodeInsertion(boolean evict) { }
```

```
// 删除结点。
void afterNodeRemoval(Node<K,V> p) { }
```

2. afterNodeInsertion调用了removeEldestEntry,若removeEldestEntry返回true 时,则会删除头结点,但是LinkedHashMap定死的,永远返回false.所以链表长度可以无限长。

3. 自定义LRUCache,重写removeEldestEntry 方法,若当前数量 > 初始容量则返回 true.

   class LRUCache extends LinkedHashMap< Integer, Integer >{ private int capacity;

```
public LRUCache1 (int capacity) {
    super(capacity, 0.75F, true);
    this.capacity = capacity;
}

public int get(int key) {
    return super.getOrDefault(key, -1);
}

public void put(int key, int value) {
    super.put(key, value);
}

@Override
protected boolean removeEldestEntry(Map.Entry<Integer, Integer>
eldest) {
    return size() > capacity;
}
```

}

dui 手写堆。小顶堆，用于升序排序。

1. 堆的两个 主要方法就是： 向上调整 up 和 向下 调整 dowm。

   1. up : 将堆末尾元素向上调整。
   2. dowm： 将堆顶元素向下调整。

2. 入队 :

   1. 若有空间，追加到堆尾部，然后 up ,向上调整。
   2. 若无空间，和堆顶比较，小于堆顶则放弃，否则，覆盖堆顶，dowm将堆顶元素向下调整

3. 出队 : 返回堆顶,然后使用堆尾,覆盖堆顶,然后dowm将堆顶元素向下调整

```
class Dui{

    int[] dui;
```

```
    int curlen;

    public Dui(int k){
        dui = new int[k];
        curlen = 0;
    }

    // 调整数组最后一个元素到合适的位置。
    public void downtoup(){
        if(curlen <= 1) return;
        int cur = curlen - 1;
        int key = dui[cur];
        while(cur > 0){
            int parent = (cur - 1) >>> 1;
            if(key >= dui[parent]) break;
            dui[cur] = dui[parent];
            // 父亲下调，儿子才能上调。
            cur = parent;
        }
        dui[cur] = key;
    }

    public void uptodown(){
        if(curlen <= 1) return;
        int cur = 0;
        int key = dui[cur];
        int half = curlen >>> 1;
        while(cur < half){
            int child = (cur << 1) + 1;
            int right = child + 1;
            if(right < curlen && dui[child] > dui[right])
                child = right;
            if(key <= dui[child]) break;
            // 孩子向上调
            dui[cur] = dui[child];
            cur = child;
        }
        dui[cur] = key;
    }

    // 向堆内添加元素
    public void offer(int key){
        // 堆还有空闲 空间
        if(curlen < dui.length){
            dui[curlen++] = key;
            // 自下向上调整。
            downtoup();
        }else{
            // 只要新元素大于 堆顶时，才会加入该元素。
            if(key > dui[0]){
                dui[0] = key;
```

```
                // 自上向下调整。
                uptodown();
            }
        }
    }

    public int poll(){
        if(curlen == 0) return Integer.MIN_VALUE;  // error
        if(curlen == 1) return dui[--curlen];
        int res = dui[0];
        dui[0] = dui[--curlen];
        uptodown();
        return res;
    }
 }
```

最小生成树的

0. 输入数据 第一行一个整数代表点数，第二行是边

   5 1,4,70;1,5,30;1,3,60;3,4,30;3,5,10;4,5,25;2,4,10 边 1,4,70 代表 第一个点和第四个点之间相连。

1. 并查集

```
 public void Unioncode() throws IOException{
    BufferedReader bf = new BufferedReader(new
InputStreamReader(System.in));
    String line_1 = bf.readLine();
    int n = Integer.parseInt(line_1);
    // 一个点，一个临接表。
    ArrayList<int[]>[] edges = new ArrayList[n];
    int[] parent = new int[n];
    for(int i = 0; i < n; i++){
        edges[i] = new ArrayList<>();
        parent[i] = i;
    }

    String[] line_2 = bf.readLine().split(";");
    // 分号隔开每一条边。
    PriorityQueue<int[]> que = new PriorityQueue<>((o1,o2)->{
        return o1[2] - o2[2];
    });

    for(String s : line_2){
        String[] edge = s.split(",");
        int from = Integer.parseInt(edge[0])-1;
        int to = Integer.parseInt(edge[1])-1;
        int w = Integer.parseInt(edge[2]);
        que.offer(new int[]{from, to, w});
```

```
        }
        // 最小生成树有 n-1条边。
        int[][] ans = new int[n-1][2];
        int ind = 0;

        while(!que.isEmpty()){
            int[] edge = que.poll();
            int x = edge[0], y = edge[1], d = edge[2];

            int xp = find(parent, x);
            int yp = find(parent, y);
            // 在同一个集合中，不能连接。
            if(xp == yp) continue;
            // 连接。
            parent[xp] = yp;
            // 记录边
            ans[ind][0] = x+1; ans[ind++][1] = y+1;
        }
        // 输出 n-1条边
        for(int i = 0; i < n -1; i++){
            System.out.println(ans[i][0] +" - to -" + ans[i][1]);
        }
    }
    public int find(int[] parent,int x){
        if(parent[x] != x){
            parent[x] = find(parent, parent[x]);
        }
        return parent[x];
    }
```

2. prim算法

```
    public void primecode() throws IOException{
        BufferedReader bf = new BufferedReader(new
InputStreamReader(System.in));
        String line_1 = bf.readLine();
        int n = Integer.parseInt(line_1);
        // 一个点，一个临接表。
        ArrayList<int[]>[] edges = new ArrayList[n];
        for(int i = 0; i < n; i++){
            edges[i] = new ArrayList<>();
        }

        String[] line_2 = bf.readLine().split(";");
        // 分号隔开每一条边。
        for(String s : line_2){
            String[] edge = s.split(",");
            int from = Integer.parseInt(edge[0])-1;
            int to = Integer.parseInt(edge[1])-1;
```

```
            int w = Integer.parseInt(edge[2]);
            edges[from].add(new int[]{to , w});
            edges[to].add(new int[]{from , w});
        }
        // 最小生成树有 n-1条边。
        int[][] ans = new int[n-1][2];
        int ind = 0;
        PriorityQueue<int[]> que = new PriorityQueue<>((o1,o2)->{
            return o1[2] - o2[2];
        });
        // 假设每次都从0开始生成最小生成树。
        int v0 = 0;
        for(int[] edge : edges[v0]){
            // 起点，终点，权重。
            que.offer(new int[]{v0,edge[0],edge[1]});
        }
        boolean[] vis = new boolean[n];
        vis[v0] = true;
        while(!que.isEmpty()){
            int[] edge = que.poll();
            // 起点，终点，权重。
            int x = edge[0], y = edge[1], d = edge[2];
            if(vis[y]) continue;
            vis[y] = true;
            ans[ind][0] = x+1; ans[ind++][1] = y+1;
            for(int[] newedge : edges[y]){
                // 只能加入to点没有访问的边。
                if(!vis[newedge[0]])
                    que.offer(new int[]{y,newedge[0],newedge[1]});
            }
        }
        // 输出 n-1条边
        for(int i = 0; i < n -1; i++){
            System.out.println(ans[i][0] +" - to -" + ans[i][1]);
        }
    }
```

最短路径

1. Dijkstras算法： 模仿 prim算法的方式即可。更新优先队列方式修改一下即可。

并查集模板

```
    public int find(int[] parent,int x){
        if(parent[x] != x){
            parent[x] = find(parent, parent[x]);
        }
        return parent[x];
    }
```

快速排序模板

```
public void sort(int[] nums, int left, int right){
    int mid = partition(nums, left, right);
    if(mid+1 < right)
        sort(nums, mid+1, right);
    if(left < mid-1)
        sort(nums, left, mid-1);
}

public int partition(int[] nums, int left, int right) {

    int pivot = nums[left];
    int i = left; int j = right;
    while (i < j) {
        while (i < j && nums[j] >= pivot) --j;
        if(i < j)  nums[i] = nums[j];

        while (i < j && nums[i] < pivot) ++i;
        if(i < j)  nums[j] = nums[i];
    }
    nums[i] = pivot;
    return i;
}
```

Hj18 : 验证 ip 和掩码

1. 正确的掩码必须满足以下条件,ip > 0 && ip < 0XFFFFFFFFL && (((ip ^ 0XFFFFFFFFL) + 1) | ip) == ip,在(0,255)之间开区间。且掩码取反+1或自己还等于自己。
2. A类ip : ip >= 1 && ip <= 126 前缀：0： 0000 0001(1) ~ 0111 1110(126) 0111 1111:127保留。
3. B类ip : ip >= 128 && ip <= 191 前缀：10： 1000 0000(128) ~ 1011 1111(191)
4. C类ip : ip >= 192 && ip <= 223 前缀: 110 : 1100 0000(192) ~ 1101 1111(223)
5. D类ip : ip >= 224 && ip <= 239 前缀: 1110 : 1110 0000(224) ~ 1110 1111(239)
6. E类ip : ip >= 240 && ip <= 255 前缀: 1111 0 : 1111 0000(240) ~ 1111 1111(255)